



Magento® U

Contents

Unit One. Preparation & Configuration	5
Module 4. Magento 2 Overview.....	5
1.4.1. Create a new module. Make a mistake in its config. Create a second module dependent on the first.	5
Module 7. Dependency Injection & Object Manager	6
1.7.1. In the empty module you created in Exercise 1.4.1, add custom configuration xml/xsd files.	6
Module 8. Plugins.....	11
1.8.1. For \Magento\Catalog\Model\Product getPriceMethod(), create a plugin and preference.	11
Module 9. Events.....	12
1.9.1. Create an observer to the event controller_action_predispatch	12
Unit Two. Request Flow	13
Module 2. Request Flow Overview.....	13
2.2.1. Find a place in the code where output is flashed to the browser. Create an extension that captures and logs the file-generated page HTML.	13
Module 3. Request Routing.....	14
2.3.1. Create an extension that logs into the file list of all available routers.	14
2.3.2. Create a new router that “understands” URLs like /frontName-actionPath-action and converts them to: /frontName/actionPath/action.	14
2.3.3. Modify Magento so a “Not Found” page will forward to the home page.	15
Module 5. Working with Controllers.....	17
2.5.1. Create a frontend controller that renders “HELLO WORLD”.	17
2.5.2. Customize the catalog product view controller using plugins and preferences.	17
2.5.3. Create an adminhtml controller that allows access only if the GET parameter “secret” is set.	18
2.5.4. Make the “Hello World” controller you just created redirect to a specific category page.	19
Module 6. URL Rewrites	19
2.6.1. Create a URL rewrite for the “Hello World” controller.	19
Unit Three. Rendering	20
Module 3. Rendering Flow	20
3.3.1. In the core files, find and print out the layout xml for the product view page.	20
Module 5. Block Architecture & Lifecycle	20
3.5.1. Create a block extending AbstractBlock, and implement the _toHtml() method. Render that block in the new controller.	20
3.5.2. Create and render in controller text block.	21
3.5.3. Customize the Catalog\Product\View\Description block, implement the _beforeToHtml() method, and set the custom description to the product here.	21
Module 6. Templates.....	22
3.6.1. Define which template is used in Catalog\Block\Product\View\Attributes.....	22
3.6.2. Create a template block and a custom template file for it. Render the block in the controller.	22
3.6.3. Customize the Catalog\Block\Product\View\Description block and set a custom template to it.	22
Module 8. Layout Loading & Rendering	23
3.8.1. Add a default.xml layout file to the Training_Render module.	23
3.8.2. Add an arguments/argument node to the block.	23
3.8.3. Change the block color to orange on the product detail pages only.	23
3.8.4. On category pages, move the exercise block to the bottom of the left column.	23
3.8.5. Create a new controller action (ex: training_render/layout/onepage).	23

3.8.6. On the custom action you just added, remove the custom block from the content.top container (see Exercise 3.8.1).	23
3.8.7. Using Layout XML, add a new link for the custom page you just created to the set of existing links at the top on every page.	23
Unit 4. Databases and Entity-Attribute-Value (EAV)	24
Module 2. Databases Overview	24
4.2.1. List Root Categories by Store	24
Module 3. Models Detailed Workflow	24
4.3.1. Product Save operations.	24
Module 4. Setup Scripts & Setup Resources	24
4.4.1. Create a table with an install script for the module Training_Vendor.	24
4.4.2. Create a regular upgrade script to add an additional column.	24
4.4.3. Create a data upgrade script to set a config value.	24
Module 5. Entity-Attribute-Value Concepts	25
4.5.1. Create a table with an install script for the module Training_Vendor.	25
Module 7. Attribute Management	25
4.7.1. Create a text input attribute from the Admin interface.	25
4.7.2. Create a text input attribute from an install data method.	26
4.7.3. Create a multi-select product attribute from an upgrade data method.	27
4.7.4. Customize the frontend rendering of the attribute values.	30
4.7.5. Create a select attribute with a predefined list of options.	32
Unit 5. Service Contracts	35
Module 4. Service API: Repositories, Search Criteria, Business Logic	35
Preparation	35
5.4.1. Obtain a list of products via the product repository.	35
5.4.2. Obtain a list of customers via the customer repository.	38
5.4.3. Create a service API and repository for a custom entity.	41
Module 5. Data and Web API	53
5.5.1. Create a new entity category_countries.	53
5.5.2. Create scripts that make SOAP calls.	53
5.5.3. Perform an API call to the "V1/customers/1" path.	53
5.5.4. Create your own Data API class and make it available through the Web API.	54

Unit One. Preparation & Configuration

Module 4. Magento 2 Overview

1.4.1. Create a new module. Make a mistake in its config. Create a second module dependent on the first.

Solution

1. Create a folder `app/code/Training/Test`.
2. Create a file `app/code/Training/Test/etc/module.xml`:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/Module/etc/module.xsd">
    <module name="Training_Test" schema_version="2.0.0">
    </module>
</config>
```

3. Add your new module to the `app/etc/config.php` in the list of modules.
4. Make a mistake in the `module.xml`. For example change `</module>` to `</mod>`. Then clean the cache (using the command `rf -rf var/cache/*`) and load any page. You should get an error:
Warning: DOMDocument::loadXML(): Opening and ending tag mismatch: module line 8 and mod in Entity, line: 9 in `/var/www/magento/m2/lib/internal/Magento/Framework/Module/ModuleList/Loader.php` on line 56.

5. Fix the xml and clean the cache again.

6. Create a folder `app/code/Training/Test2` and file `app/code/Training/Test2/etc/module.xml`:

```
<?xml version="1.0"?>
<!--
/**
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/Module/etc/module.xsd">
    <module name="Training_Test2" schema_version="2.0.0">
        <sequence>
            <module name="Magento_Test"/>
        </sequence>
    </module>
</config>
```

7. Add `Training_Test2` to the list of modules in `app/etc/config.php`.
8. Clean the cache, and test whether your module is working.
9. You can disable `Training_Test` by setting its value to 0 in the `etc/config.php`
10. After cleaning the cache, there will be no visible change. In order to see a list of loaded modules, go into the class `Magento\Framework\Module\ModuleList`, method `getNames()`, and put `print_r($result); exit;` before the return from the method. It will show a list of loaded methods, and you will see `Training_Test2` but no `Training_Test`.

Module 7. Dependency Injection & Object Manager

1.7.1. In the empty module you created in Exercise 1.4.1, add custom configuration xml/xsd files.

In order to create new xml/xsd files, we have to take the following steps:

Phase 1: Create `test.xml` and `test.xsd` files.

Phase 2: Create php files to process them: `Config`, `ConfigInterface`, `Convertor`, `Reader`, `SchemaLocator`.

Phase 3: Define a preference for `ConfigInterface`.

Phase 4: Test: In this example we will create a new controller to test this functionality out.

Let's follow through each step:

Phase 1

1.1) Create etc/test.xml:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="config.xsd">
    <mynode>HELLO</mynode>
    <mynode>HELLO 2</mynode>
</config>
```

1.2) Create etc/test.xsd:

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="config">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="mynode" type="xs:string" maxOccurs="10"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Phase 2

2.1) Create an interface:

```
<?php
namespace Training\Test\Model\Config;

interface ConfigInterface {
    public function getMyNodeInfo();
}
```

2.2) Create Config class:

```
<?php
namespace Training\Test\Model;

class Config extends \Magento\Framework\Config\Data implements
\Training\Test\Model\Config\ConfigInterface
{
    public function __construct(
        \Training\Test\Model\Config\Reader $reader,
        \Magento\Framework\Config\CacheInterface $cache,
        $cacheId = 'training_test_config'
    ) {
        parent::__construct($reader, $cache, $cacheId);
    }

    public function getMyNodeInfo() {
        return $this->get();
    }
}
```

2.3) Create Reader class:

```
<?php
namespace Training\Test\Model\Config;

class Reader extends \Magento\Framework\Config\Reader\Filesystem
{
    /**
     * List of id attributes for merge
     *
     * @var array
     */
    protected $_idAttributes = []; //['/config/option' => 'name',
'/config/option/inputType' => 'name'];

    /**
     * @param \Magento\Framework\Config\FileResolverInterface $fileResolver
     * @param \Magento\Catalog\Model\ProductOptions\Config\Converter $converter
     * @param \Magento\Catalog\Model\ProductOptions\Config\SchemaLocator
     * $schemaLocator
     * @param \Magento\Framework\Config\ValidationStateInterface $validationState
     * @param string $fileName
     * @param array $idAttributes
     * @param string $domDocumentClass
     * @param string $defaultScope
     */
}
```

```
public function __construct(
    \Magento\Framework\Config\FileResolverInterface $fileResolver,
    \Training\Test\Model\Config\Converter $converter,
    \Training\Test\Model\Config\SchemaLocator $schemaLocator,
    \Magento\Framework\Config\ValidationStateInterface $validationState,
    $fileName = 'test.xml',
    $idAttributes = [],
    $domDocumentClass = 'Magento\Framework\Config\Dom',
    $defaultScope = 'global'
) {
    parent::__construct(
        $fileResolver,
        $converter,
        $schemaLocator,
        $validationState,
        $fileName,
        $idAttributes,
        $domDocumentClass,
        $defaultScope
    );
}
```

2.4) Create schemaLocator class:

```
<?php
namespace Training\Test\Model\Config;

class SchemaLocator implements \Magento\Framework\Config\SchemaLocatorInterface
{
    /**
     * Path to corresponding XSD file with validation rules for merged config
     *
     * @var string
     */
    protected $_schema = null;

    /**
     * Path to corresponding XSD file with validation rules for separate config
     * files
     * @var string
     */
    protected $_perFileSchema = null;

    /**
     * @param \Magento\Framework\Module\Dir\Reader $moduleReader
     */
    public function __construct(\Magento\Framework\Module\Dir\Reader $moduleReader)
    {
        $etcDir = $moduleReader->getModuleDir('etc', 'Training_Test');
        $this->_schema = $etcDir . '/test.xsd';
    }
}
```



```

        $this->_perFileSchema = $etcDir . '/test.xsd';
    }

    /**
     * Get path to merged config schema
     *
     * @return string|null
     */
    public function getSchema()
    {
        return $this->_schema;
    }

    /**
     * Get path to pre file validation schema
     *
     * @return string|null
     */
    public function getPerFileSchema()
    {
        return $this->_perFileSchema;
    }
}

```

2.5) Create converter class:

```

<?php

namespace Training\Test\Model\Config;

class Converter implements \Magento\Framework\Config\ConverterInterface
{
    /**
     * Convert dom node tree to array
     *
     * @param \DOMDocument $source
     * @return array
     * @throws \InvalidArgumentException
     */
    public function convert($source)
    {
        $output = [];

        /** @var $optionNode \DOMNode */
        foreach ($source->getElementsByTagName('mynode') as $node) {
            $output[] = $node->textContent;
        }
        return $output;
    }
}

```

Phase 3

Set a preference in the `di.xml`:

```
<preference for="Training\Test\Model\Config\ConfigInterface"
type="Training\Test\Model\Config" />
```

Phase 4

4.1) Create a controller file (assuming you've set up `routes.xml` already):

```
<?php
/**
 * Product controller.
 *
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
namespace Training\Test\Controller\Action;

class Config extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $testConfig = $this->_objectManager-
>get('Training\Test\Model\Config\ConfigInterface');
        $myNodeInfo = $testConfig->getMyNodeInfo();
        if (is_array($myNodeInfo)) {
            foreach($myNodeInfo as $str) {
                $this->getResponse()->appendBody($str . "<BR>");
            }
        }
    }
}
```

4.2) Hit a page `/test/action/config`. You will see:

```
HELLO
HELLO 2
```

Module 8. Plugins

1.8.1. For \Magento\Catalog\Model\Product getPriceMethod(), create a plugin and preference.

Task 1: Create a plugin.

Task 2: Create a preference.

**** Please note, in real-world situations, you should use task 1 OR task 2, but not both.**

Solution

Task 1

1.1) Add a plugin declaration into di.xml:

```
<type name="Magento\Catalog\Model\Product">
    <plugin name="magento-catalog-product-plugin"
        type="Training\Test\Model\Product" sortOrder="10"/>
</type>
```

1.2) Create a plugin class:

```
<?php
namespace Training\Test\Model;

class Product {
    public function afterGetPrice(\Magento\Catalog\Model\Product $product, $result)
    {
        return 5;
    }
}
```

1.3) Visit any page (after cleaning the cache). You should see every price being set to \$5.

Task 2

Note: Don't forget to disable the declaration done in Option 1!

2.1) Create a preference declaration:

```
<preference for="Magento\Catalog\Model\Product"
    type="Training\Test\Model\Testproduct" />
```

2.2) Create a new Product class:

```
<?php
namespace Training\Test\Model;

class Testproduct extends \Magento\Catalog\Model\Product
{
    public function getPrice() {
        return 3;
    }
}
```

2.3) Test. Now all prices should be set to \$3.

Module 9. Events

1.9.1. Create an observer to the event controller_action_predispatch

Solution

1. Create an event declaration in the events.xml:

```
<event name="controller_action_predispatch">
    <observer name="training_test"
        instance="Training\Test\Model\Observer"
        method="changeRequestParams" shared="false" />
</event>
```

2. Create an Observer:

```
<?php
namespace Training\Test\Model;

class Observer {

    public function changeRequestParams(\Magento\Framework\Event\Observer $observer)
    {
        $request = $observer->getEvent()->getData('request');
        $request->setModuleName('catalog');
        $request->setControllerName('product');
        $request->setActionName('view');
        $request->setParams(array('id' => 1));
    }
}
```

Result: Now all pages are “Not found.”

3. Comment out `setModuleName()`, `setControllerName()`, `setActionName()` but be sure to leave `setParams()`. Now all product pages refer to the same product page.

```
public function changeRequestParams(\Magento\Framework\Event\Observer $observer)
{
    $request = $observer->getEvent()->getData('request');
    //$request->setModuleName('catalog');
    //$request->setControllerName('product');
    //$request->setActionName('view');
    $request->setParams(array('id' => 1));
}
```

Unit Two. Request Flow

Module 2. Request Flow Overview

2.2.1. Find a place in the code where output is flashed to the browser. Create an extension that captures and logs the file-generated page HTML.

Solution

1. Declare an event in the file etc/frontend/events.xml:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/Event/etc/events.xsd">
    <event name="controller_front_send_response_before">
        <observer name="training_test" instance="Training\Test\Model\Observer"
method="logPageOutput" shared="false" />
    </event>
</config>
```

2. Create an observer class:

```
<?php
namespace Training\Test\Model;

class Observer {
    protected $_logger = null;

    public function __construct(\Psr\Log\LoggerInterface $logger) {
        $this->_logger = $logger;
    }

    public function logPageOutput(\Magento\Framework\Event\Observer $observer) {
        return;
        $response = $observer->getEvent()->getData('response');
        $body = $response->getBody();
        $this->_logger->addDebug("-----\n\n\n BODY \n\n\n ". $body);
    }
}
```

Module 3. Request Routing

2.3.1. Create an extension that logs into the file list of all available routers.

Solution

1. Create a preference in the di.xml:

```
<preference for="Magento\Framework\App\FrontControllerInterface"
type="Training\Test\App\FrontController" />
```

2. Implement a front controller class:

```
<?php
namespace Training\Test\App;

class FrontController extends \Magento\Framework\App\FrontController
{
    protected $_routerList;
    protected $_logger;

    public function __construct(\Magento\Framework\App\RouterList $routerList,
\Psr\Log\LoggerInterface $logger)
    {
        $this->_routerList = $routerList;
        $this->_logger = $logger;
    }

    public function dispatch(\Magento\Framework\App\RequestInterface $request) {
        foreach ($this->_routerList as $router) {
            $this->_logger->addDebug(get_class($router));
        }
        return parent::dispatch($request);
    }
}
```

2.3.2. Create a new router that “understands” URLs like /frontName-actionPath-action and converts them to: /frontName/actionPath/action.

Solution

1. Declare your router. Add the following code to the etc/frontend/di.xml of your module (assuming your module is Training_Test):

```
<type name="Magento\Framework\App\RouterList">
    <arguments>
        <argument name="routerList" xsi:type="array">
            <item name="training" xsi:type="array">
                <item name="class"
                    xsi:type="string">Training\Test\Controller\Router</item>
                <item name="disable" xsi:type="boolean">false</item>
            </item>
        </argument>
    </arguments>
</type>
```

```

        <item name="sortOrder" xsi:type="string">70</item>
    </item>
</argument>
</arguments>
</type>

```

2. Create a router class:

```

<?php

namespace Training\Test\Controller;

class Router implements \Magento\Framework\App\RouterInterface
{
    public function __construct(\Magento\Framework\App\ActionFactory $actionFactory)
    {
        $this->actionFactory = $actionFactory;
    }

    public function match(\Magento\Framework\App\RequestInterface $request) {
        $info = $request->getPathInfo();

        if (preg_match("%^(test)-(.?)-(.?)$", $info, $m)) {
            $request->setPathInfo(sprintf("/%s/%s/%s", $m[1], $m[2], $m[3]));
            return $this->actionFactory-
                >create('Magento\Framework\App\Action\Forward', ['request' =>
                    $request]);
        }
        return null;
    }
}

```

In this example, the router only “understands” urls that start with “test”. To make it work with every url, remove the line:

```
if (preg_match("%^(test)-(.?)-(.?)$", $info, $m)) {
```

2.3.3. Modify Magento so a “Not Found” page will forward to the home page.

Solution

There are many different ways to do this. The easiest is to change the config option `/web/default/noroute`. This will change the 404 page for all requests. To make the code more flexible, you can create a new `NoRouteHandler`. To do this:

1. Declare your handler in the di.xml:

```

<type name="Magento\Framework\App\Router\NoRouteHandlerList">
    <arguments>
        <argument name="handlerClassesList" xsi:type="array">
            <item name="default" xsi:type="array">

```

```
        <item name="class"
            xsi:type="string">Training\Test\Controller\NoRouteHandler</item>
        </item>
        <item name="sortOrder" xsi:type="string">200</item>
    </argument>
</arguments>
</type>
```

2. Create a handler class:

```
<?php
namespace Training\Test\Controller;

class NoRouteHandler implements
\Magento\Framework\App\Router\NoRouteHandlerInterface {

    public function process(\Magento\Framework\App\RequestInterface $request) {
        $moduleName      = 'cms';
        $controllerName   = 'index';
        $actionName       = 'index';

        $request
            ->setModuleName($moduleName)
            ->setControllerName($controllerName)
            ->setActionName($actionName);
        return true;
    }
}
```


Module 5. Working with Controllers

2.5.1. Create a frontend controller that renders “HELLO WORLD”.

Solution

1. Declare a route in `etc/frontend/routes.xml`:

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/App/
etc/routes.xsd">
    <router id="standard">
        <route id="test" frontName="test">
            <module name="Training_Test" />
        </route>
    </router>
</config>
```

2. Create an action class:

```
<?php
/**
 * Product controller.
 *
 * @copyright Copyright (c) 2014 X.commerce, Inc. (http://www.magentocommerce.com)
 */
namespace Training\Test\Controller\Action;

class Index extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $this->getResponse()->appendBody("HELLO WORLD");
    }
}
```

2.5.2. Customize the catalog product view controller using plugins and preferences.

Solution

1. To add a plugin or preference, use the following code in `di.xml`:

```
<preference for="Magento\Catalog\Controller\Product\View"
type="Training\Test\Controller\Product\View" />
```

Or

```
<type name="Magento\Catalog\Controller\Product\View">
    <plugin name="product-view-controller-plugin"
        type="Training\Test\Controller\Product\View" sortOrder="10"/>
</type>
```

Note: You will create a preference OR plugin within one module.

2. Now you can implement your preference/plugin:

```
<?php

namespace Training\Test\Controller\Product;

class View extends \Magento\Framework\App\Action\Action
{
    /**
     * public function execute() {
     *     echo "ONE"; exit;
     * }
     * public function beforeExecute() {
     *     //echo "BEFORE<BR>"; exit;
     * }
     * public function afterExecute(\Magento\Catalog\Controller\Product\View
    $controller, $result) {
     *     //echo "AFTER"; exit;
     * }
     */
}
```

3. Uncomment the appropriate method for testing: Uncomment “execute” for preferences, and “beforeExecute”, “afterExecute” for plugins.

2.5.3. Create an adminhtml controller that allows access only if the GET parameter “secret” is set.

Solution

1. Create a file Etc/adminhtml/routes.xml:

```
<?xml version="1.0"?>
<!--
/**
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/App/
etc/routes.xsd">
    <router id="admin">
        <route id="test" frontName="test">
            <module name="Training_Test" before="Magento_Adminhtml" />
        </route>
    </router>
</config>
```

2. Create an action class:

```

<?php
/**
 *
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
namespace Training\Test\Controller\Adminhtml\Action;

class Index extends \Magento\Backend\App\Action
{
    /**
     * Product list page
     *
     * @return \Magento\Backend\Model\View\Result\Page
     */
    public function execute()
    {
        $this->getResponse()->appendBody("Hello world in admin");
    }

    protected function _isAllowed() {
        $secret = $this->getRequest()->getParam('secret');
        return isset($secret) && (int)$secret==1;
    }
}

```

2.5.4. Make the “Hello World” controller you just created redirect to a specific category page.

Solution

Put a line `$this->_redirect('catalog/category/view/id/_CATEGORY_ID_')` into the execute method (but replace `_CATEGORY_ID_` with the real category_id).

Module 6. URL Rewrites

2.6.1. Create a URL rewrite for the “Hello World” controller.

Solution

Add one record to the url_rewrite table:

```

INSERT INTO url_rewrite SET request_path='testpage.html',
target_path='test/action/index', redirect_type=0, store_id=1, is_autogenerated=0;

```

Unit Three. Rendering

Module 3. Rendering Flow

3.3.1. In the core files, find and print out the layout xml for the product view page.

Solution

```
\Magento\Framework\View\Layout::generateXml()
```

Module 5. Block Architecture & Lifecycle

3.5.1. Create a block extending AbstractBlock, and implement the _toHtml() method. Render that block in the new controller.

Solution

1. Create the block:

```
<?php
namespace Training\Test\Block;

class Test extends \Magento\Framework\View\Element\AbstractBlock
{
    protected function _toHtml() {
        return "<b>Hello world from block!</b>";
    }
}
```

2. Create an action class:

```
<?php
namespace Training\Test\Controller\Block;

class Index extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $layout = $this->_view->getLayout();
        $block = $layout->createBlock('Training\Test\Block\Test');
        $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.5.2. Create and render in controller text block.

Solution

Create an action class:

```
<?php

namespace Training\Test\Controller\Block;

class Text extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $block = $this->_view->getLayout()-
>createBlock('Magento\Framework\View\Element\Text');
        $block->setText("Hello world from text block !");
        $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.5.3. Customize the Catalog\Product\View\Description block, implement the `_beforeToHtml()` method, and set the custom description to the product here.

Solution

1. Declare a plugin in the `etc/frontend/di.xml`:

```
<type name="Magento\Catalog\Block\Product\View\Description">
    <plugin name="product-view-description-plugin"
        type="Training\Test\Block\Product\View\Description" sortOrder="10"/>
</type>
```

2. Create a plugin class:

```
<?php
namespace Training\Test\Block\Product\View;

class Description extends \Magento\Framework\View\Element\Template
{
    public function beforeToHtml(\Magento\Catalog\Block\Product\View\Description
$originalBlock) {
        $originalBlock->getProduct()->setDescription('Test description');
    }
}
```

Module 6. Templates

3.6.1. Define which template is used in Catalog\Block\Product\View\Attributes.

Solution

Magento/Catalog/view/frontend/templates/product/view/attributes.phtml

3.6.2. Create a template block and a custom template file for it. Render the block in the controller.

Solution

1. Create the block:

```
<?php
namespace Training\Test\Block;

class Template extends \Magento\Framework\View\Element\Template
{
}
```

Note: You cannot create your own block. You must use Magento\Framework\View\Element\Template, since it is not an abstract.

2. Create a template file Training/Test/view/frontend/test.phtml:

"Hello from template".

3. Create an action class:

```
<?php
namespace Training\Test\Controller\Block;

class Template extends \Magento\Framework\App\Action\Action
{
    public function execute() {
        $block = $this->_view->getLayout()-
>createBlock('Training\Test\Block\Template');
        $block->setTemplate('test.phtml');
        $this->getResponse()->appendBody($block->toHtml());
    }
}
```

3.6.3. Customize the Catalog\Block\Product\View\Description block and set a custom template to it.

Solution

1. Using the same declaration as in 2.3, change the beforeToHtml method to:

```
public function beforeToHtml(\Magento\Catalog\Block\Product\View\Description
$originalBlock) {
    $originalBlock->setTemplate('Training_Test::description.phtml');
}
```

2. Create a template `Training/Test/view/frontend/templates/description.phtml`:

```
<h1>Custom description template!</h1>
```

Module 8. Layout Loading & Rendering

You will be provided with a code archive containing the solutions for the exercises in this module.

3.8.1. Add a `default.xml` layout file to the `Training_Render` module.

1. Reference the `content.top` container.
2. Add a `Magento\Framework\View\Element\Template` block with a custom template.
3. Create your custom template.
4. Check that the template content is visible on every page.

3.8.2. Add an `arguments/argument` node to the block.

1. Set the argument name to `background_color`.
2. Set the argument value to `lightskyblue..`
3. In the template, add an inline style attribute to a `<div>` element:
4. `style="background_color:`
5. `<?= $this->getData('background_color') ?>;"`
6. Check that the background color is displayed.

3.8.3. Change the block color to orange on the product detail pages only.

3.8.4. On category pages, move the exercise block to the bottom of the left column.

3.8.5. Create a new controller action (ex: `training_render/layout/onepage`).

- For that action, choose a single column page layout using layout XML.
- Set a page title using layout XML.

3.8.6. On the custom action you just added, remove the custom block from the `content.top` container (see Exercise 3.8.1).

3.8.7. Using Layout XML, add a new link for the custom page you just created to the set of existing links at the top on every page.

Unit Four. Databases and Entity-Attribute-Value (EAV)

Module 2. Databases Overview

4.2.1. List Root Categories by Store

Solution

You will be provided with a code archive containing the solutions for the exercises in this module.

Module 3. Models Detailed Workflow

4.3.1. Product Save operations.

- Log every Product Save operation.
- Specify the productId and the data that has been changed.

Module 4. Setup Scripts & Setup Resources

4.4.1. Create a table with an install script for the module Training_Vendor.

1. Give Training_Vendor a setup_version of 0.0.1.
2. Create the Setup folder.
3. Create the InstallSchema class.
4. Create a training_vendor_entity table using DDL methods.
5. Execute the installation using the console tool.
6. Verify that it works by checking the setup_module table.

4.4.2. Create a regular upgrade script to add an additional column.

1. Create the UpgradeSchema class.
2. Add an additional column to the training_vendor_entity table using DDL adapter methods.
3. Upgrade the version number in your module.xml to 0.0.2.
4. Run the appropriate console command.
5. Verify that it works.

4.4.3. Create a data upgrade script to set a config value.

1. Create the UpgradeData class.
2. Define a fixture vendor to be installed along with your module.
3. Upgrade the module version.
4. Execute the appropriate console command.
5. Verify that it works.

You will be provided with a code archive containing the solutions for the exercises in this module.

Module 5. Entity-Attribute-Value Concepts

4.5.1. Create a table with an install script for the module Training_Vendor.


1. Give Training_Vendor a setup_version of 0.0.1.
2. Create the Setup folder.
3. Create the InstallSchema class.
4. Create a training_vendor_entity table using DDL methods.
5. Execute the installation using the console tool.
6. Verify that it works by checking the setup_module table.

Module 7. Attribute Management

4.7.1. Create a text input attribute from the Admin interface.

1. Add it to an attribute set.
2. Check that it appears on the product edit page.
3. Make it visible on the storefront product view page.

Solution

 Note that this exercise does not require any coding. It must be completed using the browser to access the Magento Admin backend and the storefront only.

1. Log in to the Magento backend.
2. Select **Stores > Attribute > Product** in the main navigation.
3. Click the **Add New Attribute** button.
4. Enter an attribute label, for example: **Flavor**.
5. Select the tab **Frontend Properties**.
6. Set **Visible on Catalog Pages on Frontend** to **Yes**.
7. Click **Save Attribute**.
8. Select **Stores > Attribute > Product Templates** in the main navigation.
9. In the list, select an Attribute Set, for example the **Bag** attribute set.
10. Drag & drop the **Flavor** attribute icon from the right **Unassigned Attributes** column onto the **Product Details** attribute group folder icon.
11. Confirm that the **Flavor** attribute icon now is listed within the **Product Details** attribute group.
12. Click **Save Attribute Set**.
13. Select **Products > Inventory > Catalog** in the main navigation.
14. In the **Attribute Set** column filter dropdown, select **Default** and click the **Search** button.
15. Select a product from the list where the **Visibility** is set to **Catalog, Search**, for example the **Push It Messenger Bag**.
16. Confirm that the **Flavor** attribute field is displayed on the **Product Details** form.
17. Enter a value for the **Flavor** attribute, for example **Strawberry**.
18. Click the **Save** button.
19. Open the product in the Magento storefront.
20. Select the **Additional Information** tab.
21. Confirm that the new attribute and the value you gave it are displayed.

4.7.2. Create a text input attribute from an install data method.

Follow the steps in the previous exercise and create a text input attribute that is visible on the storefront, but this time create it from an install data method.

Solution

The exercise solution requires coding.

1. Create a module Training_Orm.
2. Create an InstallData setup class.

```
<?php
```

```
namespace Training\Orm\Setup;

use Magento\Catalog\Model\Product;
use Magento\Catalog\Model\Resource\Eav\Attribute as CatalogAttribute;
use Magento\Catalog\Setup\CategorySetup;
use Magento\Catalog\Setup\CategorySetupFactory;
use Magento\Framework\Setup\InstallDataInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;

class InstallData implements InstallDataInterface
{
    /**
     * @var CategorySetupFactory
     */
    private $catalogSetupFactory;

    public function __construct(CategorySetupFactory $categorySetupFactory)
    {
        $this->catalogSetupFactory = $categorySetupFactory;
    }

    /**
     * Installs data for a module
     *
     * @param ModuleDataSetupInterface $setup
     * @param ModuleContextInterface $context
     * @return void
     */
    public function install(
        ModuleDataSetupInterface $setup,
        ModuleContextInterface $context
    ) {
    }
}
```

3. In the install method, create an instance of the CategorySetup class and call addAttribute on it.

```
public function install(
    ModuleDataSetupInterface $setup,
    ModuleContextInterface $context
) {
    /** @var CategorySetup $catalogSetup */
    $catalogSetup = $this->catalogSetupFactory->create(['setup' => $setup]);

    $catalogSetup->addAttribute(Product::ENTITY, 'flavor_from_setup_method', [
        'label' => 'Flavor From Setup Method',
        'visible_on_front' => 1,
        'required' => 0,
        'global' => CatalogAttribute::SCOPE_STORE
    ]);
}
```

4. Apply the setup changes.

```
% ./bin/magento setup:upgrade
Cache cleared successfully
File system cleanup:
.../var/generation/Magento
Updating modules:
Schema creation/updates:
...
Module 'Training_Orm':
Installing data..
```

5. Open a product in the backend interface and confirm that the new attribute exists and can be set on a store view level.
6. Visit a product on the storefront and confirm that the new attribute is visible there, too.

4.7.3. Create a multi-select product attribute from an upgrade data method.

- Create a multi-select product attribute.
- Set the backend_model property to Magento\Eav\EntityAttribute\Backend\Array.
- Add a few options to the attribute.
- Make it visible in the catalog product view page.

Solution

The exercise solution requires coding.

1. Expand the Training_Orm module by adding an UpgradeData setup class.

<?php

```
namespace Training\Orm\Setup;

use Magento\Catalog\Model\Product;
use Magento\Catalog\Setup\CategorySetup;
use Magento\Eav\Model\Entity\Attribute\Backend\ArrayBackend;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;
use Magento\Framework\Setup\UpgradeDataInterface;
use Magento\Catalog\Setup\CategorySetupFactory;
use Magento\Store\Model\StoreManagerInterface as StoreManager;

class UpgradeData implements UpgradeDataInterface
{
    /**
     * @var CategorySetupFactory
     */
    private $catalogSetupFactory;

    /**
     * @var StoreManager
     */
    private $storeManager;

    public function __construct(
        CategorySetupFactory $categorySetupFactory,
        StoreManager $storeManager
    ) {
        $this->catalogSetupFactory = $categorySetupFactory;
        $this->storeManager = $storeManager;
    }

    /**
     * Upgrades data for a module
     *
     * @param ModuleDataSetupInterface $setup
     * @param ModuleContextInterface $context
     * @return void
     */
    public function upgrade(
```

```

        ModuleDataSetupInterface $setup,
        ModuleContextInterface $context
    ) {
    }
}

```

2. Add an upgrade script to add the new attribute.

```

public function upgrade(
    ModuleDataSetupInterface $setup,
    ModuleContextInterface $context
) {
    $dbVersion = $context->getVersion();

    if (version_compare($dbVersion, '0.1.1', '<')) {
        $admin = $this->storeManager->getStore('admin')->getId();
        $default = $this->storeManager->getStore('default')->getId();

        /** @var CategorySetup $catalogSetup */
        $catalogSetup = $this->catalogSetupFactory->create(['setup' => $setup]);
        $catalogSetup->addAttribute(Product::ENTITY, 'example_multiselect', [
            'label' => 'Example Multi-Select',
            'input' => 'multiselect',
            'backend' => ArrayBackend::class,
            'visible_on_front' => 1,
            'required' => 0,
            'option' => [
                'order' => ['option1' => 10, 'option2' => 20],
                'value' => [
                    'option1' => [
                        $admin => 'Admin Label 1',
                        $default => 'Frontend Label 1'
                    ],
                    'option2' => [
                        $admin => 'Admin Label 2',
                        $default => 'Frontend Label 2'
                    ]
                ]
            ]
        ]);
    }
}

```

3. Update the setup version in the module.xml file to 0.1.1.

```

<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/Module/etc/modu

```

```
le.xsd">
    <module name="Training_Orm" setup_version="0.1.1">
        <sequence>
            <module name="Magento_Eav"/>
        </sequence>
    </module>
</config>
```

4. Run the upgrade script processing.

```
% ./bin/magento setup:upgrade
Cache cleared successfully
File system cleanup:
.../var/generation/Magento
Updating modules:
Schema creation/updates:
...
Module 'Training_Orm':
Installing data..
```

5. Open a product in the backend interface and confirm that the new attribute exists, and that both options are available.
6. Visit a product on the storefront and confirm that the new attribute is visible there, too.

4.7.4. Customize the frontend rendering of the attribute values.

- Customize the rendering of the values of the multi-select product attribute that you created in the previous exercise.
- Show it as an HTML list rather than as comma-separated values.

Solution

The exercise solution requires coding.

1. Expand the Training_Orm module by adding a new section to the UpgradeData setup class.

```
public function upgrade(
    ModuleDataSetupInterface $setup,
    ModuleContextInterface $context
) {
    $dbVersion = $context->getVersion();

    if (version_compare($dbVersion, '0.1.1', '<')) {
        // ...
    }
    if (version_compare($dbVersion, '0.1.2', '<')) {
        /** @var CategorySetup $catalogSetup */
        $catalogSetup = $this->catalogSetupFactory->create(['setup' => $setup]);
        $catalogSetup->updateAttribute(
            Product::ENTITY,
```

```

        'example_multiselect',
        [
            'frontend_model' =>
                \Training\Orm\Entity\Attribute\Frontend\HtmlList::class,
            'is_html_allowed_on_front' => 1,
        ]
    );
}
}

```

2. Update the setup version in the module.xml file to 0.1.2.

```

<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/Module/etc/module.xsd">
    <module name="Training_Orm" setup_version="0.1.2">
        <sequence>
            <module name="Magento_Eav"/>
        </sequence>
    </module>
</config>

```

3. Create the HtmlList frontend model.

```

<?php

namespace Training\Orm\Entity\Attribute\Frontend;

class HtmlList extends
    \Magento\Eav\Model\Entity\Attribute\Frontend\AbstractFrontend
{
    /**
     * @param \Magento\Framework\Object $object
     * @return string
     */
    public function getValue(\Magento\Framework\Object $object)
    {
        if ($this->getConfigField('input') !== 'multiselect') {
            return parent::getValue($object);
        }
        return $this->getValuesAsHtmlList($object);
    }

    /**
     * @param \Magento\Framework\Object $object
     * @return string
     */
}

```

```
private function getValuesAsHtmlList(\Magento\Framework\Object $object)
{
    $options = $this->getOptions($object);
    $escapedOptions = array_map('htmlspecialchars', $options);
    return sprintf(
        '<ul><li>%s</li></ul>',
        implode('</li><li>', $escapedOptions)
    );
}

/**
 * @param \Magento\Framework\Object $object
 * @return string[]
 */
private function getOptions(\Magento\Framework\Object $object)
{
    $optionId = $object->getData($this->getAttribute()->getAttributeCode());
    $option = $this->getOption($optionId);
    return $this->isSingleValue($option) ? [$option] : $option;
}

/**
 * @param string[]|string $option
 * @return bool
 */
private function isSingleValue($option)
{
    return !is_array($option);
}
}
```

4. Run the setup upgrade process.
5. Set both options on a product and save it.
6. View the product on the storefront and confirm the option values are rendered as a HTML list.

4.7.5. Create a select attribute with a predefined list of options.

- Create a new customer attribute 'priority' using an upgrade data method.
- Use the frontend_input type 'select'.
- Use the backend_type 'int'.
- Set is_system to 0.
- Assign a custom source model.
- Implement the custom attribute source model to list numbers from 1 through 10.
- Test that the attribute works as expected.

Solution

The exercise solution requires coding.

1. Expand the Training_Orm module by adding a new dependency on Magento\Customer\Setup\CustomerSetupFactory to the DataUpgrade class. Initialize the field using dependency injection.

```
// ...
use Magento\Customer\Setup\CustomerSetup;
use Magento\Customer\Setup\CustomerSetupFactory;

// ...

public function __construct(
    CategorySetupFactory $categorySetupFactory,
    CustomerSetupFactory $customerSetupFactory,
    StoreManager $storeManager
) {
    $this->catalogSetupFactory = $categorySetupFactory;
    $this->customerSetupFactory = $customerSetupFactory;
    $this->storeManager = $storeManager;
}
// ...
```

2. Add a new section to the UpgradeData setup class.

```
public function upgrade(
    ModuleDataSetupInterface $setup,
    ModuleContextInterface $context
) {
    $dbVersion = $context->getVersion();

    if (version_compare($dbVersion, '0.1.1', '<')) {
        // ...
    }
    if (version_compare($dbVersion, '0.1.2', '<')) {
        // ...
    }
    if (version_compare($dbVersion, '0.1.3', '<')) {
        /** @var CustomerSetup $customerSetup */
        $customerSetup = $this->customerSetupFactory->create(['setup' => $setup]);
        $customerSetup->addAttribute(
            Customer::ENTITY,
            'priority',
            [
                'label' => 'Priority',
                'type' => 'int',
                'input' => 'select',
                'source' => \Training\Orm\Entity\Attribute\Source\CustomerPriority::class,
                'required' => 0,
                'system' => 0,
                'position' => 100
            ]
        );
    }
}
```

```
    );  
    $customerSetup->getEavConfig()->getAttribute('customer', 'priority')  
        ->setData('used_in_forms', ['adminhtml_customer'])  
        ->save();  
    }  
}
```

3. Note that the system property must be set to 0 for Magento to recognize the custom attribute when saving a customer through the Admin interface.
4. Note that customer attributes must be added to the customer_form_attribute explicitly to be visible in the browser.
5. Create the custom source model.

<?php

```
namespace Training\Orm\Entity\Attribute\Source;  
  
use Magento\Eav\Model\Entity\Attribute\Source\AbstractSource;  
  
class CustomerPriority extends AbstractSource  
{  
    /**  
     * Retrieve All options  
     *  
     * @return array[]  
     */  
    public function getAllOptions()  
    {  
        $options = array_map(function($priority) {  
            return [  
                'label' => sprintf('Priority %d', $priority),  
                'value' => $priority  
            ];  
        }, range(1, 10));  
        if ($this->getAttribute()->getFrontendInput() === 'select') {  
            array_unshift($options, ['label' => '', 'value' => 0]);  
        }  
        return $options;  
    }  
}
```

6. Run the setup scripts.
7. Visit a customer in the Admin interface, and select the **Account Information** tab to confirm that the field is visible and that the list of priorities is visible.
8. Set a value for the Priority attribute and save the customer to confirm that the value is being saved on the entity.

Unit Five. Service Contracts

Module 4. Service API: Repositories, Search Criteria, Business Logic

Use the native customer and product repository classes to obtain lists of objects.

Preparation

1. Create a new module Training_Registry for the exercise solutions.
2. Add a frontend route configuration.

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/App/etc/routes.xsd">
    <router id="standard">
        <route id="training_repository" frontName="training_repository">
            <module name="Training_Registry"/>
        </route>
    </router>
</config>
```

5.4.1. Obtain a list of products via the product repository.

- Print a list of products.
- Add a filter to the search criteria.
- Add another filter with a logical AND condition.
- Add a sort order instruction.
- Limit the number of products to 6.

Solution

1. Create an action controller to output the exercise result.

```
<?php
```

```
namespace Training\Repository\Controller\Repository;
```

```
use Magento\Catalog\Api\Data\ProductInterface;
use Magento\Catalog\Api\ProductRepositoryInterface;
use Magento\Framework\Api\SearchCriteriaBuilder;
use Magento\Framework\App\Action\Action;
use Magento\Framework\App\Action\Context;
```

```
class Product extends Action
{
    /**
     * @var ProductRepositoryInterface
     */
```

```
private $productRepository;

/**
 * @var SearchCriteriaBuilder
 */
private $searchCriteriaBuilder;

public function __construct(
    Context $context,
    ProductRepositoryInterface $productRepository,
    SearchCriteriaBuilder $searchCriteriaBuilder
) {
    parent::__construct($context);
    $this->productRepository = $productRepository;
    $this->searchCriteriaBuilder = $searchCriteriaBuilder;
}

public function execute()
{
    $this->getResponse()->setHeader('Content-Type', 'text/plain');

    $products = $this->getProductsFromRepository();

    foreach ($products as $product) {
        $this->outputProduct($product);
    }
}

/**
 * @return ProductInterface[]
 */
private function getProductsFromRepository()
{
    $criteria = $this->searchCriteriaBuilder->create();
    $products = $this->productRepository->getList($criteria);
    return $products->getItems();
}

private function outputProduct(ProductInterface $product)
{
    $this->getResponse()->appendBody(sprintf(
        "%s - %s (%d)\n",
        $product->getName(),
        $product->getSku(),
        $product->getId()
    ));
}
}
```

If there is no output when testing the action in this stage, check the PHP error logs for out-of-memory exceptions.

2. Add a filter so the result list contains only configurable products.

```
public function __construct(
    Context $context,
    ProductRepositoryInterface $productRepository,
    SearchCriteriaBuilder $searchCriteriaBuilder,
    FilterBuilder $filterBuilder
) {
    parent::__construct($context);
    $this->productRepository = $productRepository;
    $this->searchCriteriaBuilder = $searchCriteriaBuilder;
    $this->filterBuilder = $filterBuilder;
}

// ... the execute() method is unchanged ...

/**
 * @return ProductInterface[]
 */
private function getProductsFromRepository()
{
    $this->setProductTypeFilter();

    $criteria = $this->searchCriteriaBuilder->create();
    $products = $this->productRepository->getList($criteria);
    return $products->getItems();
}

private function setProductTypeFilter()
{
    $configProductFilter = $this->filterBuilder
        ->setField('type_id')
        ->setValue(ConfigurableProduct::TYPE_CODE)
        ->setConditionType('eq')
        ->create();
    $this->searchCriteriaBuilder->addFilter([$configProductFilter]);
}
```

 Note that the imports have to be adjusted accordingly.

3. Add another filter that is applied using a logical AND operator by adding the following method and calling it from the `getProductsFromRepository()` method.

```
private function setProductNameFilter()
{
    $nameFilter = $this->filterBuilder
        ->setField('name')
        ->setValue('M%')
        ->setConditionType('like')
        ->create();
}
```

```
$this->searchCriteriaBuilder->addFilter([$nameFilter]);
}
```

4. Add a sort order instruction by adding the SortOrderBuilder and SearchCriteriaInterface to the class dependencies and adding the following method (called from the getProductsFromRepository() method).

```
private function setProductOrder()
{
    $sortOrder = $this->sortOrderBuilder
        ->setField('entity_id')
        ->setDirection(SearchCriteriaInterface::SORT_ASC)
        ->create();
    $this->searchCriteriaBuilder->addSortOrder($sortOrder);
}
```

5. Limit the number of product to 6. This can be done by renaming the new method setProductOrder() to setProductPaging() and adding the new 2 lines at the end of the method.

```
private function setProductPaging()
{
    $sortOrder = $this->sortOrderBuilder
        ->setField('entity_id')
        ->setDirection(SearchCriteriaInterface::SORT_ASC)
        ->create();
    $this->searchCriteriaBuilder->addSortOrder($sortOrder);
    $this->searchCriteriaBuilder->setPageSize(6);
    $this->searchCriteriaBuilder->setCurrentPage(1);
}
```

5.4.2. Obtain a list of customers via the customer repository.

- Output the object type.
- Print a list of customers.
- Add a filter to the search criteria.
- Add another filter with a logical OR condition.

Solution

1. Add a new action controller, Customer.php, to the training_repository/repository route created in the previous exercise.
2. In the execute method of the controller, use the customer repository to get a list of customers and print some data.

```
public function execute()
{
    $this->getResponse()->setHeader('content-type', 'text/plain');

    $customers = $this->getCustomersFromRepository();

    $this->getResponse()->appendBody(
        sprintf("List contains %s\n\n", get_class($customers[0]))
    );
}
```

```

    );
    foreach ($customers as $customer) {
        $this->outputCustomer($customer);
    }
}

/**
 * @return \Magento\Customer\Api\Data\CustomerInterface[]
 */
private function getCustomersFromRepository()
{
    $criteria = $this->searchCriteriaBuilder->create();
    $customers = $this->customerRepository->getList($criteria);
    return $customers->getItems();
}

private function outputCustomer(
    \Magento\Customer\Api\Data\CustomerInterface $customer
) {
    $this->getResponse()->appendBody(sprintf(
        "\"%s %s\" <%s> (%s)\n",
        $customer->getFirstname(),
        $customer->getLastname(),
        $customer->getEmail(),
        $customer->getId()
    ));
}

```

3. Output the type of the objects returned by the repository.

```

public function execute()
{
    $this->getResponse()->setHeader('content-type', 'text/plain');

    $customers = $this->getCustomersFromRepository();

    $this->getResponse()->appendBody(
        sprintf("List contains %s\n\n", get_class($customers[0]))
    );
    foreach ($customers as $customer) {
        $this->outputCustomer($customer);
    }
}

```

4. Add two filters with a logical OR condition by specifying them as a filter group.

```

namespace Training\Repository\Controller\Repository;

use Magento\Customer\Api\CustomerRepositoryInterface;
use Magento\Framework\Api\FilterBuilder;
use Magento\Framework\Api\Search\FilterGroupBuilder;
use Magento\Framework\Api\SearchCriteriaBuilder;
use Magento\Framework\App\Action\Action;

```

```
use Magento\Framework\App\Action\Context;

class Customer extends Action
{
    /**
     * @var CustomerRepositoryInterface
     */
    private $customerRepository;

    /**
     * @var SearchCriteriaBuilder
     */
    private $searchCriteriaBuilder;

    /**
     * @var FilterGroupBuilder
     */
    private $filterGroupBuilder;

    /**
     * @var FilterBuilder
     */
    private $filterBuilder;

    public function __construct(
        Context $context,
        CustomerRepositoryInterface $customerRepository,
        SearchCriteriaBuilder $searchCriteriaBuilder,
        FilterGroupBuilder $filterGroupBuilder,
        FilterBuilder $filterBuilder
    ) {
        parent::__construct($context);
        $this->customerRepository = $customerRepository;
        $this->searchCriteriaBuilder = $searchCriteriaBuilder;
        $this->filterGroupBuilder = $filterGroupBuilder;
        $this->filterBuilder = $filterBuilder;
    }

    public function execute()
    {
        $this->getResponse()->setHeader('content-type', 'text/plain');

        $this->addEmailFilter();
        $this->addNameFilter();
        $customers = $this->getCustomersFromRepository();

        $this->getResponse()->appendBody(
            sprintf("List contains %s\n\n", get_class($customers[0]))
        );

        foreach ($customers as $customer) {
            $this->outputCustomer($customer);
        }
    }
}
```



```

private function addEmailFilter()
{
    $emailFilter = $this->filterBuilder
        ->setField('email')
        ->setValue('%@dmail.com')
        ->setConditionType('like')
        ->create();
    $this->filterGroupBuilder->addFilter($emailFilter);
}

private function addNameFilter()
{
    $nameFilter = $this->filterBuilder
        ->setField('firstname')
        ->setValue('Hans')
        ->setConditionType('eq')
        ->create();
    $this->filterGroupBuilder->addFilter($nameFilter);
}

/**
 * @return \Magento\Customer\Api\Data\CustomerInterface[]
 */
private function getCustomersFromRepository()
{
    $this->searchCriteriaBuilder->setFilterGroups(
        [$this->filterGroupBuilder->create()]
    );
    $criteria = $this->searchCriteriaBuilder->create();
    $customers = $this->customerRepository->getList($criteria);
    return $customers->getItems();
}

private function outputCustomer(
    \Magento\Customer\Api\Data\CustomerInterface $customer
) {
    $this->getResponse()->appendBody(sprintf(
        "\"%s %s\" <s> (%s)\n",
        $customer->getFirstname(),
        $customer->getLastname(),
        $customer->getEmail(),
        $customer->getId()
    ));
}
}

```

5.4.3. Create a service API and repository for a custom entity.

- Try to follow best practices.
- The custom example entity should use a flat table for storage.
- The repository only needs to contain a `getList()` method.

Solution

1. Create a new flat table entity called Example with a model, resource model, and collection. Refer to the ORM unit of the training or the exercise solution code archive for details. The model, resource model, and collection code is boilerplate and is not included in this document.

```
Repository
├── Model
│   ├── Example.php
│   └── Resource
│       ├── Example
│       │   └── Collection.php
│       └── Example.php
└── Setup
    ├── InstallSchema.php
    └── UpgradeData.php
```

2. Use a module install class to create the matching table, and a module data upgrade class to create a couple of example records in the table.

```
<?php
```

```
namespace Training\Repository\Setup;
```

```
use Magento\Framework\DB\Adapter\AdapterInterface;
use Magento\Framework\DB\Ddl\Table as DdlTable;
use Magento\Framework\Setup\InstallSchemaInterface;
use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\SchemaSetupInterface;
```

```
class InstallSchema implements InstallSchemaInterface
{
    public function install(
        SchemaSetupInterface $setup,
        ModuleContextInterface $context
    ) {
        $setup->startSetup();
        $tableName = $setup->getTable('training_repository_example');
        $ddlTable = $setup->getConnection()->newTable(
            $tableName
        );
        $ddlTable->addColumn(
            'example_id',
            DdlTable::TYPE_INTEGER,
            null,
            [
                'identity' => true,
                'unsigned' => true,
                'nullable' => false,
                'primary' => true
            ]
        )->addColumn(
```

```

        'name',
        DdlTable::TYPE_TEXT,
        255,
        ['nullable' => false]
    )->addColumn(
        'created_at',
        DdlTable::TYPE_TIMESTAMP,
        null,
        ['nullable' => false, 'default' => DdlTable::TIMESTAMP_INIT]
    )->addColumn(
        'updated_at',
        DdlTable::TYPE_TIMESTAMP,
        null,
        ['nullable' => false, 'default' => DdlTable::TIMESTAMP_INIT]
    )->addIndex(
        $setup->getIdxName(
            $tableName,
            ['name'],
            AdapterInterface::INDEX_TYPE_UNIQUE
        ),
        ['name'],
        ['type' => AdapterInterface::INDEX_TYPE_UNIQUE]
    );

    $setup->getConnection()->createTable($ddlTable);

    $setup->endSetup();
}
}

```

<?php

```

namespace Training\Repository\Setup;

use Magento\Framework\Setup\ModuleContextInterface;
use Magento\Framework\Setup\ModuleDataSetupInterface;
use Magento\Framework\Setup\UpgradeDataInterface;

class UpgradeData implements UpgradeDataInterface
{
    public function upgrade(
        ModuleDataSetupInterface $setup,
        ModuleContextInterface $context
    ) {
        $dbVersion = $context->getVersion();

        if (version_compare($dbVersion, '0.1.1', '<')) {
            $tableName = $setup->getTable('training_repository_example');
            $setup->getConnection()->insertMultiple(
                $tableName,
                [
                    ['name' => 'Foo'],
                    ['name' => 'Bar'],
                    ['name' => 'Baz'],
                    ['name' => 'Qux'],
                ]
            );
        }
    }
}

```

```
    ]  
    );  
    }  
}
```

3. Add an interface `Training\Repository\Api\ExampleRepositoryInterface`. It contains only the `getList()` method. No framework interface needs to be extended.

```
<?php
```

```
namespace Training\Repository\Api;  
  
use Magento\Framework\Api\SearchCriteriaInterface;  
  
interface ExampleRepositoryInterface  
{  
    /**  
     * @return Data\ExampleSearchResultsInterface  
     */  
    public function getList(SearchCriteriaInterface $searchCriteria);  
}
```

4. Add an interface for the API data model `Training\Repository\Api\Data\ExampleInterface` with getters and setters for all the properties that should be accessible from the outside.

```
<?php
```

```
namespace Training\Repository\Api\Data;  
  
interface ExampleInterface  
{  
    /**  
     * @param int $id  
     * @return $this  
     */  
    public function setId($id);  
  
    /**  
     * @return int  
     */  
    public function getId();  
  
    /**  
     * @return string  
     */  
}
```

```

public function getName();

/**
 * @param string $name
 * @return $this
 */
public function setName($name);

/**
 * @return string
 */
public function getCreatedAt();

/**
 * @param string $createdAt
 * @return $this
 */
public function setCreatedAt($createdAt);

/**
 * @return string
 */
public function getModifiedAt();

/**
 * @param string $modifiedAt
 * @return $this
 */
public function setModifiedAt($modifiedAt);
}

```

5. Add an interface `Training\Repository\Api\Data\ExampleSearchResultsInterface` that extends `Magento\Framework\Api\SearchResultsInterface`.

It can inherit all methods, or specify `getItems()` and `setItems()` to provide more specific phpdoc type hints.

<?php

```

namespace Training\Repository\Api\Data;

interface ExampleSearchResultsInterface
    extends \Magento\Framework\Api\SearchResultsInterface
{
    /**
     * @api
     * @return \Training\Repository\Api\Data\ExampleInterface[]
     */
}

```

```
public function.getItems();

/**
 * @api
 * @param \Training\Repository\Api\Data\ExampleInterface[] $items
 * @return $this
 */
public function setItems(array $items = null);
}
```

6. Specify the preferences configuration for these three new interfaces in an etc/di.xml file.

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework/ObjectManager/etc/config.xsd">
    <preference for="Training\Repository\Api\ExampleRepositoryInterface"
                type="Training\Repository\Model\ExampleRepository"/>
    <preference for="Training\Repository\Api\Data\ExampleInterface"
                type="Training\Repository\Model\Example"/>
    <preference for="Training\Repository\Api\Data\ExampleSearchResultsInterface"
                type="Magento\Framework\Api\SearchResults"/>
</config>
```

7. Make the Training\Repository\Model\Example class implement the Api\Data\ExampleInterface.

```
<?php

namespace Training\Repository\Model;

use Magento\Framework\Model\AbstractModel;
use Training\Repository\Api\Data\ExampleInterface;

class Example extends AbstractModel implements ExampleInterface
{
    protected function _construct()
    {
        $this->_init(Resource\Example::class);
    }

    public function getName()
    {
        return $this->_getData('name');
    }
}
```

```

public function setName($name)
{
    $this->setData('name', $name);
}

public function getCreatedAt()
{
    return $this->_getData('created_at');
}

public function setCreatedAt($createdAt)
{
    $this->setData('modified_at', $createdAt);
}

public function getModifiedAt()
{
    return $this->_getData('modified_at');
}

public function setModifiedAt($modifiedAt)
{
    $this->setData('modified_at', $modifiedAt);
}
}

```

8. Finally it is time to create the repository implementation.

The `getList()` method creates an example collection instance and applies the `SearchCriteria` using the appropriate methods on the collection.

Then the collection is loaded, and all entities are converted into the configured object implementation for the `Api\Data\ExampleInterface`.

For this implementation it means that the conversion happens to the same instance, but should the DI configuration for the interface change, this ensures that change will take effect.

<?php

```
namespace Training\Repository\Model;
```

```

use Magento\Framework\Api\Search\FilterGroup;
use Magento\Framework\Api\SearchCriteriaInterface;
use Training\Repository\Api\Data\ExampleInterface;
use Training\Repository\Api\Data\ExampleInterfaceFactory as ExampleDataFactory;
use Training\Repository\Api\Data\ExampleSearchResultsInterface;
use Training\Repository\Api\Data\ExampleSearchResultsInterfaceFactory;
use Training\Repository\Api\ExampleRepositoryInterface;
use Training\Repository\Model\Example as ExampleModel;
use Training\Repository\Model\Resource\Example\Collection as ExampleCollection;

```

```
class ExampleRepository implements ExampleRepositoryInterface
{
    /**
     * @var ExampleSearchResultsInterfaceFactory
     */
    private $searchResultsFactory;

    /**
     * @var ExampleFactory
     */
    private $exampleFactory;

    /**
     * @var ExampleDataFactory
     */
    private $exampleDataFactory;

    public function __construct(
        ExampleSearchResultsInterfaceFactory $searchResultsFactory,
        ExampleFactory $exampleFactory,
        ExampleDataFactory $exampleDataFactory
    ) {
        $this->searchResultsFactory = $searchResultsFactory;
        $this->exampleFactory = $exampleFactory;
        $this->exampleDataFactory = $exampleDataFactory;
    }

    /**
     * @return ExampleSearchResultsInterface
     */
    public function getList(SearchCriteriaInterface $searchCriteria)
    {
        /** @var ExampleCollection $collection */
        $collection = $this->exampleFactory->create()->getCollection();

        /** @var ExampleSearchResultsInterface $searchResults */
        $searchResults = $this->searchResultsFactory->create();
        $searchResults->setSearchCriteria($searchCriteria);

        $this->applySearchCriteriaToCollection($searchCriteria, $collection);
        $examples = $this->convertCollectionToDataItemsArray($collection);

        $searchResults->setTotalCount($collection->getSize());
        $searchResults->setItems($examples);

        return $searchResults;
    }
}
```



```

    private function addFilterGroupToCollection(
        FilterGroup $filterGroup,
        ExampleCollection $collection
    ) {
        $fields = [];
        $conditions = [];
        foreach ($filterGroup->getFilters() as $filter) {
            $condition = $filter->getConditionType() ?
                $filter->getConditionType() :
                'eq';
            $fields[] = $filter->getField();
            $conditions[] = [$condition => $filter->getValue()];
        }
        if ($fields) {
            $collection->addFieldToFilter($fields, $conditions);
        }
    }

    private function convertCollectionToDataItemsArray(
        ExampleCollection $collection
    ) {
        $examples = array_map(function (ExampleModel $example) {
            /** @var ExampleInterface $dataObject */
            $dataObject = $this->exampleDataFactory->create();
            $dataObject->setId($example->getId());
            $dataObject->setName($example->getName());
            $dataObject->setCreatedAt($example->getCreatedAt());
            $dataObject->setModifiedAt($example->getModifiedAt());
            return $dataObject;
        }, $collection->getItems());
        return $examples;
    }

    private function applySearchCriteriaToCollection(
        SearchCriteriaInterface $searchCriteria,
        ExampleCollection $collection
    ) {
        $this->applySearchCriteriaFiltersToCollection(
            $searchCriteria,
            $collection
        );
        $this->applySearchCriteriaSortOrdersToCollection(
            $searchCriteria,
            $collection
        );
        $this->applySearchCriteriaPagingToCollection(
            $searchCriteria,
            $collection
        );
    }
}

```

```
private function applySearchCriteriaFiltersToCollection(
    SearchCriteriaInterface $searchCriteria,
    ExampleCollection $collection
) {
    foreach ($searchCriteria->getFilterGroups() as $group) {
        $this->addFilterGroupToCollection($group, $collection);
    }
}

private function applySearchCriteriaSortOrdersToCollection(
    SearchCriteriaInterface $searchCriteria,
    ExampleCollection $collection
) {
    $sortOrders = $searchCriteria->getSortOrders();
    if ($sortOrders) {
        $isAscending =
            $sortOrder->getDirection() == SearchCriteriaInterface::SORT_ASC;
        foreach ($sortOrders as $sortOrder) {
            $collection->addOrder(
                $sortOrder->getField(),
                $isAscending ? 'ASC' : 'DESC'
            );
        }
    }
}

private function applySearchCriteriaPagingToCollection(
    SearchCriteriaInterface $searchCriteria,
    ExampleCollection $collection
) {
    $collection->setCurPage($searchCriteria->getCurrentPage());
    $collection->setPageSize($searchCriteria->getPageSize());
}
}
```

9. Create an action controller to test the result.

```
<?php
```

```
namespace Training\Repository\Controller\Repository;

use Magento\Framework\Api\FilterBuilder;
use Magento\Framework\Api\SearchCriteriaBuilder;
use Magento\Framework\App\Action\Action;
use Magento\Framework\App\Action\Context;
use Training\Repository\Api\ExampleRepositoryInterface;
```

```

class Example extends Action
{
    /**
     * @var ExampleRepositoryInterface
     */
    private $exampleRepository;

    /**
     * @var SearchCriteriaBuilder
     */
    private $searchCriteriaBuilder;

    /**
     * @var FilterBuilder
     */
    private $filterBuilder;

    public function __construct(
        Context $context,
        ExampleRepositoryInterface $exampleRepository,
        SearchCriteriaBuilder $searchCriteriaBuilder,
        FilterBuilder $filterBuilder
    ) {
        $this->exampleRepository = $exampleRepository;
        $this->searchCriteriaBuilder = $searchCriteriaBuilder;
        $this->filterBuilder = $filterBuilder;
        parent::__construct($context);
    }

    public function execute()
    {
        $this->getResponse()->setHeader('content-type', 'text/plain');

        $filters = array_map(function ($name) {
            return $this->filterBuilder
                ->setConditionType('eq')
                ->setField('name')
                ->setValue($name)
                ->create();
        }, ['Foo', 'Bar', 'Baz', 'Qux']);
        $this->searchCriteriaBuilder->addFilter($filters);
        $examples = $this->exampleRepository->getList(
            $this->searchCriteriaBuilder->create()
        )->getItems();
    }
}

```

```
foreach ($examples as $example) {  
    $this->getResponse()->appendBody(sprintf(  
        "%s (%d)\n",  
        $example->getName(),  
        $example->getId()  
    ));  
}  
  
}
```

Module 5. Data and Web API

5.5.1. Create a new entity category_countries.

- Include category_country_id, category_id, country_id.
- Add a few records to that table (using DataInstallScript).
- Add an extension attribute "countries" to the category.

Solution

You will be provided with a code archive containing the solutions for the exercises in this module.

5.5.2. Create scripts that make SOAP calls.

- Create a php-script that performs a SOAP call to the customer repository getById() method.
- Create a php-script that performs a SOAP call to the customer repository getList() method. Define the filter & sorting options in the SearchCriteria parameter.
- Create a php-script that performs a SOAP call to the catalog product repository getList() method.
- Add a new attribute in the Admin and make a SOAP call to the catalog product repository get() method to obtain a product with a list of attributes. Make sure your new attribute is there.

Solution

5.5.3. Perform an API call to the "V1/customers/1" path.

- Explore the Magento_Customer module and find other examples of the available services. Perform a call to some service you've found.

Solution

```
<?php

require('vendor/zendframework/zend-server/Zend/Server/Client.php');
require('vendor/zendframework/zend-soap/Zend/Soap/Client.php');
require('vendor/zendframework/zend-soap/Zend/Soap/Client/Common.php');

$wsdlUrl = 'http://magento.loc/m2-0.74/soap/V1/?wsdl&services=customerAccountManagementV1,
customerCustomerRepositoryV1';

$token = '12345';

$options = ['http' => ['header' => "Authorization: Bearer " . $token]];
$context = stream_context_create($options);

$soapClient = new \Zend\Soap\Client($wsdlUrl);
$soapClient->setSoapVersion(SOAP_1_2);
$soapClient->setStreamContext($context);
$result = $soapClient->customerAccountManagementV1IsReadonly(array('customerId' => 1));
var_dump($result);
```

5.5.4. Create your own Data API class and make it available through the Web API.

1. In order to perform a REST API call to the Customer module, perform the following two http calls:

a. `curl -X POST "http://magento.loc/m2-0.74/index.php/rest/V1/integration/admin/token" -H "Content-Type:application/json" -d '{"username": "_ADMIN_USERNAME_", "password": "_ADMIN_PASSWORD_"}'`

It will return a token.

b. `curl -X GET "http://magento.loc/m2-0.74/index.php/rest/V1/customers/1" -H "Authorization: Bearer _ADMIN_TOKEN_"`

Where `_ADMIN_TOKEN_` is the one from the previous call.

2. Create a module Api:

module.xml

```
<?xml version="1.0"?>
<!--
/**
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework\
Module/etc/module.xsd">
    <module name="Training_Api" setup_version="1.0.0">
        </module>
    </config>
```

di.xml

```
<?xml version="1.0"?>
<!--
/**
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../../../../lib/internal/Magento/Framework\
ObjectManager/etc/config.xsd">
```

```

        <preference for="Training\Api\Api\Data\HelloInterface" type="Training\Api\Model\Hello"
    />
</config>

```

webapi.xml

```

<?xml version="1.0"?>
<!--
/**
 * Copyright © 2015 Magento. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="../../../app/code/Magento/Webapi/etc/webapi.xsd">

    <route url="/V1/traininghello" method="GET">
        <service class="Training\Api\Api\Data\HelloInterface" method="sayHello"/>
        <resources>
            <resource ref="anonymous" />
        </resources>
    </route>
</routes>

```

Api/Api/Data/HelloInterface.php

```

<?php

namespace Training\Api\Api\Data;

interface HelloInterface {

    /**
     * Hello method
     *
     * @return string|null
     */
    public function sayHello();
}

```

Api/Model/Hello.php

```

<?php

namespace Training\Api\Model;

use Training\Api\Api\Data;

class Hello implements \Training\Api\Api\Data\HelloInterface {

```

```
public function sayHello() {  
    return "HELLO WORLD!";  
}  
  
}
```

Call it by hitting a URL: <http://magento.loc/m2-0.74/index.php/rest/V1/traininghello>

You can do this either by using curl or using a browser.

The result should be:

```
<?xml version="1.0"?>  
<response>HELLO WORLD!</response>
```